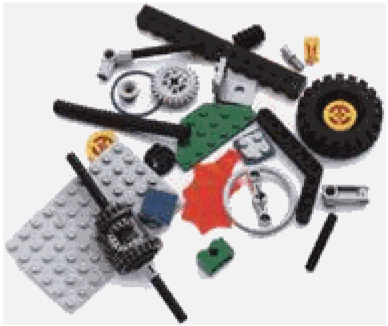
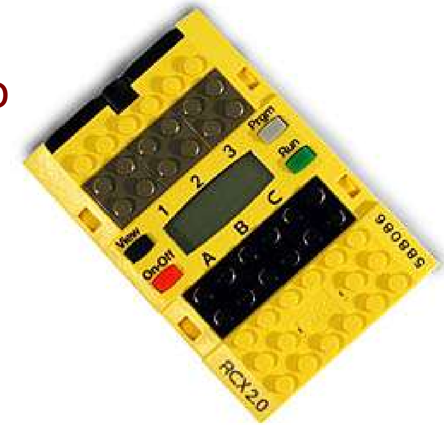


California State University, Chico
Intelligent Systems Laboratory
Chico, CA 95929-0410
http://www.gotbots.org



LEGO Mindstorms RIS 2.0
Programming: NQC Code

B.A. Juliano and R.S. Renner

September 2004

Back Forward

NQC

- NQC is short for "Not Quite C"
  - Written by Dave Baum
  - Text-based language
  - Based on C programming language, but specialized for robots, and simpler than full C
    - Syntax is similar to C++ and Java
    - More flexible than Lego RCX Code – better for intermediate and higher level programmers ...

Back Forward

GUIs for NQC

- RcxCC: RCX Command Center
  - http://www.cs.uu.nl/people/markov/lego/rcxcc/
- BricxCC
  - A GUI for using NQC in Windows environment
  - Written by Mark Overmars
  - http://bricxcc.sourceforge.net/
- MacNQC:
  - A GUI for using NQC in Mac environment
  - Written by K. Robert Bate
  - http://homepage.mac.com/rbate/MacNQC/

Back Forward

Simple example of NQC code

```
task main()
{
    SetSensor(SENSOR_1, SENSOR_LIGHT);
    On(OUT_A + OUT_C);
    while(true)
    {
        if (SENSOR_1 < 43)
        {
            SetDirection(OUT_A + OUT_C, OUT_FWD);
        }
        else
        {
            SetDirection(OUT_A + OUT_C, OUT_REV);
        }
    }
}
```

NQC Code vs RCX Code …

- Every RCX program starts with a program block
- On/Off blocks refer to output ports A, B, C
- Sensor blocks assign input ports to sensors, and associate actions with sensor readings

- Every NQC program contains a block “task main()”
- Output ports are called OUT_A, OUT_B, OUT_C
- Control output with statements OnFwd(), OnRev(), Off(), etc.
- Input ports are called SENSOR_1 (or 2 or 3)

What does this program do?

```
task main()
{
  SetPower
  (OUT_A+OUT_C,2);
  OnFwd(OUT_A+OUT_C);
  Wait(400);
  OnRev(OUT_A+OUT_C);
  Wait(400);
  Off(OUT_A+OUT_C);
}
```

Back Forward

What about this program?

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  repeat(4)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    OnRev(OUT_C);
    Wait(TURN_TIME);
  }
  Off(OUT_A+OUT_C);
}
```

- #define
  - Preprocessor directive – use before task main()
  - Used to define ‘macros’, i.e., simple name substitutions that cannot be changed in the program
  - Here it is used to define constants
- repeat() { … }
  - A control structure that alters the usual sequential execution
  - Permits a block of statements to be repeated a specified number of times.

Back Forward

Nesting and Comments

```
/*  10 SQUARES by Mark Overmars
    This program make the robot run 10 squares
*/

#define MOVE_TIME   100      // Time for a straight move
#define TURN_TIME    85      // Time for turning 90 degrees

task main()
{
  repeat(10)                 // Make 10 squares
  {
    repeat(4)
    {
      OnFwd(OUT_A+OUT_C);
      Wait(MOVE_TIME);
      OnRev(OUT_C);
      Wait(TURN_TIME);
    }
  }
  Off(OUT_A+OUT_C);          // Now turn the motors off
}
```

Variables

- A constant is a named value that cannot be changed.
  - `#define MOVE_TIME 100`
- A **variable** is a named value that can be changed
  - You must first declare the variable:
    - `int a; //declare variable named 'a'`
    - `int b = 37; //declare and initialize variable 'b'`
  - You declare each variable only once
  - If you declare inside a task, the variable only exists inside that task (local variable)
  - If you declare outside any task, the variable exists for all tasks (global variable)

Back Forward

Arithmetic Operations

- The code at right illustrates some arithmetic operations:
  - =
    - assignment of a value to a variable
  - +, -, *, /
    - Usual arithmetic operators
  - ++, --
    - Increment (add 1), decrement (subtract 1)
  - +=, -=, *=, /=
    - Add (subtract, multiply, divide) value on right to current value of variable on left;
    - int n = 10;
      n *= 3; // n is now 10* 3 = 30
- Trace the code at right and give the final values of the variables aaa, bbb, and ccc. Say which are local and which are global

```
int aaa = 10;
int bbb;
task main()
{
   int ccc;
   aaa = 10;
   bbb = 20 * 5;
   ccc = bbb;
   ccc += aaa;
   ccc /= 5;
   aaa = 10 * (ccc + 3);

   ++aaa;
}
```

The function Random()

- Random(n)
  - An expression equal to a random value from 0 to n (inclusive)
  - Changes each time it is executes
- What does the code at right do?

```
int move_time, turn_time;

task main()
{
  while(true)
  {
    move_time = Random(600);
    turn_time = Random(40);
    OnFwd(OUT_A+OUT_C);
    Wait(move_time);
    OnRev(OUT_A);
    Wait(turn_time);
  }
}
```

Control Structures

- A **control structure** is any statement that alters the order in which other statements are executed.
- NQC **decision** control structures:
  - if (condition) {…}
  - if (condition) {…} else {…}
- NQC **iteration** (repetition) control structures
  - repeat (expression) {…}
  - while (condition) {…}
  - do (condition) {…}
  - until (condition) {…}

Boolean (true/false) operators

==	equal to (different from =, which is assignment)
<	smaller than
<=	smaller than or equal to
>	larger than
>=	larger than or equal to
!=	not equal to

true	always true
false	never true
ttt != 3	true when ttt is not equal to 3

(ttt >= 5) && (ttt <= 10)
true when ttt lies between 5 and 10

(aaa == 10) || (bbb == 10)
true if either aaa or bbb (or both) are equal to 10

Back Forward

Example

```
#define MOVE_TIME   100
#define TURN_TIME    85

task main()
{
  while(true)
  {
    OnFwd(OUT_A+OUT_C);
    Wait(MOVE_TIME);
    if (Random(1) == 0)
    {
      OnRev(OUT_C);
    }
    else
    {
      OnRev(OUT_A);
    }
    Wait(TURN_TIME);
  }
}
```

- What does code at left do?
- Classic == v. = error:
  - ```
    int n = 0;
    until (n = 10)
    {
        PlaySound(1);
    }
    ```
    ```

Using Sensors

- To use a sensor, we
  - 1. Assign it to an input port
    - SetSensor(SENSOR_1, SENSOR_TOUCH);
  - 2. Choose actions based on its values:
    - **if** (SENSOR_1 == 1) {…}
    - A sensor in a program is like a constant – it has a value that you cannot change in the program (but its value is changed by the physical sensor readings from the input port)

Example

- Here is some line following code:

```
#define THRESHOLD 40

task main()
{
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  OnFwd(OUT_A+OUT_C);
  while (true)
  {
    if (SENSOR_2 > THRESHOLD)
    {
      OnRev(OUT_C);
      until (SENSOR_2 <= THRESHOLD);
      OnFwd(OUT_A+OUT_C);
    }
  }
}
```

Tasks & Event-Driven Programming

- Each task consists of a set of statements that are executed sequentially
- The RCX can run up to 10 tasks concurrently:
  - As we know, there must be at least one task, named **main()**
  - We typically use multiple tasks so that RCX can be doing something (moving, making sounds) while at the same time getting information from sensors
- Event-driven programming:
  - Programming in which program statements are executed in response to events (sensor readings, mouse clicks or movements, etc.)
  - Event-driven program is often parallel – checking for several events, and responding, all at the same time

Back Forward

Syntax for tasks

- Each task has its own name
- The only task automatically started is task main()
  - Other tasks are started with the **start** statement, and stopped with the **stop** statement (Note: no parenthesis after task name)

```
task main()
{
  SetSensor(SENSOR_1,
     SENSOR_TOUCH);
  start check_sensors;
  start move_square;
}
```

```
task move_square()
{
  while (true)
  {
   OnFwd(OUT_A+OUT_C);
   Wait(100);
   OnRev(OUT_C);
   Wait(85);
  }
}
```

```
task check_sensors()
{
   while (true)
   {
     if (SENSOR_1 == 1)
     {
       stop move_square;
       OnRev(OUT_A+OUT_C);
       Wait(50);
       OnFwd(OUT_A);
       Wait(85);
       start move_square;
     }
   }
}
```

Back Forward

Using tasks

- Advice:
  - Always ask if you really need another task
  - Never permit two tasks to do something (move, make sounds) at the same time, to prevent conflicts
    - Whenever one task is doing something, first stop the other tasks

got bots?
http://isl.ecst.csuchico.edu/

OpenOffice.org Source Project

Modularity in programming

- **Modularity:** Writing programs by creating small blocks of code, then putting them together to form a larger block (module)
- Advantages of modularity:
  - **Readability:** Easier to read several small blocks of code than one large one
  - **Testability:** Can test each module individually, making it easier to find and fix errors.
  - **Reusability:** Can use existing modules to build new programs that are more complex

Back Forward

Subroutines, inline fns & macros

- In NQC a module is essentially a block of code that is given its own name. In NQC there are three types of modules:
  - Subroutines
  - Inline functions
  - Macros
- Each has advantages and disadvantages

Back Forward

Subroutines, inline fns & macros

- Subroutine syntax:
  - Named using word **sub:** sub turn_around();
  - Invoked by just using name: turn_around();
- Inline function syntax:
  - Named using word **void:** void turn_around();
  - Invoked by just using name: turn_around();
- Macro syntax:
  - Named on one line using word **#define:** **#define turn_around OnRev(OUT_C); Wait(340);OnFwd(OUT_A+OUT_C);**
  - Invoked by just using name (without parentheses): turn_around;

Subroutines: Pros and Cons

- At most 8 subroutines may be used
- Code stored only once in RCX, regardless of how many times subroutine is called
- Cannot call one subroutine from another subroutine
- No parameters permitted (variables inside parentheses)
- Advice: For technical reasons, do not call subroutines from different tasks

Back Forward

Inline functions: Pros and Cons

- Multiple copies in RCX memory: one for each time it's called
- No limit on number of inline functions
- OK to call from different tasks
- Inline functions can have **parameters** (in definition) and **arguments** (in call):
- Advice: Generally prefer inline functions over subroutines, unless limited memory in RCX is a problem

```
void turn_around(int
   turntime)
{ OnRev(OUT_C); Wait
   (turntime);
   OnFwd(OUT_A+OUT_C);
}
task main()
{OnFwd(OUT_A+OUT_C);
   Wait(100);
   turn_around(200);
   Wait(200);
   turn_around(50);
   Wait(100);
   turn_around(300);
   Off(OUT_A+OUT_C);
}
```

got bots? http://isl.ecst.csuchico.edu/

OpenOffice.org Source Project

Back Forward

Macros: Pros and Cons

- Must define on a single line
- Multiple copies, one for each call
- Can use parameters/arguments

```
#define forwards(s,t)    SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A+OUT_C);Wait(t);
#define turn_right(s,t)  SetPower(OUT_A+OUT_C,s);OnFwd(OUT_A);OnRev(OUT_C);Wait(t);

task main()
{  forwards(3,200);
   turn_right(7,85);
   forwards(7,100);
   turn_right(7,85);
   forwards(3,200);
   Off(OUT_A+OUT_C);
}
```

file:///D:/Desktop/ferdinand.gif

Marcos Macro

Back Forward

Modularity Example

- Example: Suppose we want to build a ‘line sweeper’ robot, that follows a line and, if it detects an obstruction, uses a sweeper arm to push it off the line.
- ‘Top-Down’ design: Design this first as a collection of smaller tasks (non-technical meaning) and build each one using inline functions
  - Note: We’ll often use the word ‘subroutine’ generically, to mean either a subroutine, inline function, or macro
  - Ferrari avoids use of multiple tasks when possible

'Line sweeper' top-down design

- Program outline uses subroutines to carry out individual jobs
- We can write and test each of these separately
- If we change the physical design, we can easily change one subroutine
  - **Initialize():** Assigns ports to sensors
  - **Go_Straight():** Starts motion
  - **Check_Bumper():** detects and deals with
  - **Follow_Line():** moves forward, keeping to line

```
int floor = 45;
int line = 35;

task main()
{
    Initialize();
    Go_Straight();
    while(true)
    {
        Check_Bumper();
        Follow_Line();
    }
}
```

Back Forward

Inline funtions

- **Format of inline function (subroutine) definitions:**
  - **void FunctionName( )**
    {
    list of statements
    }
- **Appears outside task main()**
  - **We'll put them after task main()**
- **Function is invoked by using name as statement:**
  - **Initialize();**

```
void Initialize()
{
    SetSensor(SENSOR_1,SENSOR_TOUCH);
    SetSensor(SENSOR_2,SENSOR_LIGHT);
}
```

```
void Check_Bumper()
{
    if (SENSOR_1==1)
    {
        Stop();
        Remove_Obstacle();
        Go_Straight();
    }
}
```

Follow_Line( )

- Sample code (simplified) for a subroutine to follow the left edge of a black line.

```
void Follow_Line()
{
   if (SENSOR_2<=floor + 5)
   {  Turn_Right();
   }
   else if (SENSOR_2>=line - 5)
   {  Turn_Left();
   }
   else
   {  Go_Straight();
   }
}
```

```
void Go_Straight()
{    OnFwd(OUT_A+OUT_C);
}

void Turn_Left()
{    Off(OUT_A);
     OnFwd(OUT_C);
}
```

Back Forward

Remove_Obstacle( )

- Sample code (simplified) for a subroutine to remove an obstacle with an arm.

```
void Remove_Obstacle()
{
    OnFwd(OUT_B);
    Wait(200);
    OnRev(OUT_B);
    Wait(200);
    Off(OUT_B);
}
```

Back Forward

References

- Dean, Alice M. CS 102B: Robot Design, http://www.skidmore.edu/~adean/CS102B0409/
- InSciTE: Innovations in Science and Technology Education, www.HighTechKids.org
- LEGO.com Mindstorms Home, mindstorms.lego.com